# Jinja template variable inside variable

I'm trying to iterate a dictionary in the Jinja2 model (Ansible). One of the arrays or keys in the dictionary is 'abcd' This {{ item.value.abcd.port }} works fine, but the key abcd varies in the dictionary. I intend to do something below using variable nginx_dir. {% set nginx_dir = item.value.keys().1 %} {% set my_port = item.value.nginx_dir.port %} Or without using any variable, something like this {{ item.value.[ item.value.keys().1].port }} This documentation section covers variable visibility Jinja activity. Jinja has several in her sights. The sight is like a new transparent foil in a foil study. You can only type into the most remote foil, but read them all because you can look through them. If you remove the top foil, all data on foil will be lost. Some Jinja tags add a new layer to the stack. Currently, these are a block, such as a macro and a filter. This means that variables and other elements specified within a macro, loop, or some of the other identifiers listed above are available only in that block. Here's an example: {% macro angryhello name %} {% set angryname = name |upper %} Hello {{ name }}. Hey, name! HELLO {{ angry name }}!!!!!! 111 {% endmacro %} The angry name of the variable is only inside, not outside, the macro. The specified macros appear as variables in the context. As a result, they are also affected by the report. The macro specified within the macro is available only for the two macros (the macro itself and the macro in which it is defined). There is a specific threat to the sample code outside the visible blocks of the submodels. This code runs before the layout template code. Therefore, it can be used to forward values back to the layout template or to import macros from templates for renderings. This type of code can print data, but it does not appear as a final rendering. So no extra space will contaminate the model. Because this code runs before the actual layout template code, it is possible that the layout code will override some of these variables. In general, this is not a problem because of the names of different variables, but it can be a problem if you plan to set default values. In this case, you must test if the variable has not been specified before it is specified: {% unless page_title %} {% set page_title = 'Default Page Title' %} {% endif %} Of course, you can also use the |default filter. Explanation This template is stored in .html: <title>{{ title|default('Untitled') }}</title><body>{% block body %} {% endblock %} ... and this submodel saved as b.html: {% expands 'a.html' %} {% include 'macros.tmpl' %} {% set title = 'My Page' %} {% block body %}{{ wrap(42) }}{% endblock %} ... and this code in macros.tmpl: {% macro wrapping(text) %} [{{ text }}] {% endmacro %} .. becomes someone with the same semantics as this one (only because the value is not stored in the variable): {% filter capture ('captured', true) %} makron rivitys (teksti) %} [{{ teksti }}] {% endmacro %} {% set title='My Page' </body> </body> {% end filter %} <title>{{ title|default('Untitled') }}</title><body> {{ wrap(42) }} </body> Note This implementation was improved in Jinja 1.1. Jinja 1.0 blocks that were not top-level blocks were not applied to the layout template. Therefore, it was impossible to use conditional expressions to include non-parent templates. If you have already worked with a python, you probably know that undefined variables evoke an exception. This is different in Jinja. There is a special value called Undefined, which represents values that do not exist. Depending on the configuration, it behaves differently. To check whether a value has been specified, you can use the specified test: {{ myvariable is not specified }} returns true if the variable does not exist. SilentUndefined: Silent Undefined is the default action. An undefined object works completely differently from variables that you may know. If you print it using {{ variable }} it will not appear because it is literally empty. If you try to iterate it, it'll work. But no items are returned. If you want to check if a value is specified, you can use the specified test: There are also some additional rules associated with this special value. All mathematical operators (+, -, *, /) return the operand: {{ undefined + foo }} returns foo {{ undefined - 42 }} returns 42. Note: no -42! In any expression that is not specified, the value is unresent. It has no length, all attribute calls return undefined, and the call: {{ undefined.(attribute).attribute_too[42] }} continues to return the value undefinated. ComplainingUndefined: Starting with Jinja 1.1, it is possible to replace an undefined default object with different values. Another common undefined object that accompanies jinja is the ComplainingUndefined object. It raises exceptions as soon as you either render it or want to iterate it or try to use attributes, etc. See 249 Star 7.4k Fork 1.3k You cannot perform this activity at this time. You are signed in with another tab or window. Update session download again. You are logged off on another tab or window. Update session download again. We use optional third-party analytics cookies to understand how you use GitHub.com to build better products. Learn more. We use optional third-party analytics cookies to understand how you use GitHub.com to build better products. You can always update your selection by clicking Cookie Settings at the bottom of the page. For more information, please see our Privacy Policy. We use essential cookies to perform essential website functions, such as logging in to them. Read more Always active We use analytics cookies to understand how you use our websites to improve them, they are used to collect information about the pages you visit and how many clicks you need to complete the task. For more information, see this page about the Jinja template module. Although the most detailed description of the model language can be found in the Jinja documentation, some basic concepts are in this article. Code creation principle Code creation is based on a small script that iterates over a domain model and writes files in python jinja template language. Because the generator script has read and parsed the IDL file as a domain model, the latter is then used as the main object for code creation within the template language. Below is an example of code passing through a domain model: {% system.modules %} module for {%- module.interfaces -%} service, {{module}} interface. {{interface}} {% endfor -%} {% modules for schema.structs -%} STRUCT , {{module}}. {{struct}} {% endfor -%} {%- module.enums file -%} ENUM , {{module}}. {{enume0}} {% endfor -%} {% End %} The template iterates over domain objects and creates text written to the file. Laguage forms a Synopsis A model that contains variables and/or expressions that are replaced by values when the model is rendered. and tags that control the logic of the model. There are a few different demarcations. The default Jinja delimiters are defined as: {% ... %} for expressions {{ ... }} for expressions printed on model output {# ... #} for comments that are not included in the template output # ... ## for line statement auditing structures The control structure refers to all the things that control the flow of the program - conditional (i.e. if/elif/others), loops, and macros and blocks, for example. The default syntax displays control structures within {% ... %} blocks. Loop for each item in a row. For example, to display a list of users of an invited variable, follow these steps: <h1>Members</h1><ul> {% for users %} <li>{{ user.username|e }}</li> {% out of %} </ul> Because variables in models retain object properties, it is possible to iterate the properties of containers, such as dictation, on: <dl> {% for key, value my_dict.iteritems() %} <dt>{{ key|e }}</dt><dd>{{ value|e }}</dd> {% endfor %} There are some specific variables available inside loop block</dl> variabledescription loop.indexSilmukan current iteration. (from 1) the current iteration of loop.index0Silmuka. (from 0) loop.reindex Number of iterations of the loop.reindex Loop head (from 1) loop.revindex0Seller end (from 0) loop.firstTrue if first iteration. loop.lastTrue if last iteration. loop.length Number of items in the series. See more Jinja documentation Unlike Python, it is not possible to break or continue in a loop. However, the sequence can be filtered during iteration, allowing items to be ignored. The following example ignores all hidden users: {% for users if the user is not a user.hidden %}<li>{{ user.username|e }}</li> {% end %} The advantage is that the special loop variable is calculated correctly; therefore, users who are not iterated are not counted. If iteration didn't happen because the sequence was empty or filtering all items in the series, the default block can be rendered using a different one: <ul> {% for users %} <li>{{ user.username|e }}</li> {% other %} <li><em>users could not be found</em></li> {% out of %} </ul> Other blocks run in Python each time the corresponding loop did not break. However, since the Jinja loops cannot break, a slightly different behavior with another keyword was chosen. Loops can also be recursively used. This is useful when working with recursive data, such as sitemaps or RDFa. In order to recursively use loops, you must in principle add a recursive converter to the loop definition and call the loop variable with a new iteration where recursion is required. The following example implements a sitemap with recursive loops: <ul class=sitemap> {% sitemap> recursive %} <li><a href={{ item.href|e }}>{{ item.title }}</a> {%- if item.children -%} <ul class=submenu>{{ loop(item.children) }}</ul> {%- endif %}</li> end of {%- %} </ul> Loop variable always refers to the nearest (inning) loop. If there is more than one layer of loops, we can bind the variable loop by typing {% set outer_loop = loop %} after the loop that we want to use recursively. Then we can call it {{ outer_loop(...) }} Note that loop reservations are cleared at the end of iteration and cannot live longer than the loop area. Older versions of Jinja2 had a bug, where in some circumstances it seemed that the tasks would work. This is not supported. If the Jos statement in Jinja is comparable to a Python if statement. The simplest form can be used to test whether a variable is specified, neither empty nor unrestrue: {% if users %} <ul> {% for users %} <li>{{ user.username|e }}</li> {% end %} </ul> {% endif %} For multiple branches, eliff and more can be used as in Python. Even there, you can use more complex expressions: {% if Kenny.sick %} Kenny is sick. {% elif kenny.dead %} You killed Kenny! You son of a bitch!!! {% other %} Kenny seems --- so far {% endif %} Tests In addition to filters, so-called {% endif%} tests are also available. The tests can be used to test a variable against a common expression. Testing a variable or expression uses its name, and then uses the test name. For example, to determine whether a variable has been specified, you can try a name that returns true or unreal, based on whether the name is specified in the current model context. Even tests can accept arguments. If the test has only one argument, parentheses can be omitted. For example, the following two expressions do the same: {% if loop.index is a 3%} {% if loop.index is a shareable(3) %} list of Builtin tests can be found on the Jinja documentation page. Filters Variables can be modified into filters Functions. Filters are separated from the variable by a pipe heart (|), and parentheses can have optional arguments. Multiple filters can be chained. The result of one filter is applied to another. 1000 1000 {{ name|striptags|title }} removes all HTML tags from the variable name and labels the output (title(striptags(name)). Filters that accept arguments have parentheses around the argument, just as they do in a function call. For example: {{ listx|join(', ') }} joins the list with commas (match str.join(', 'listx)). However, the variable filter that is applied is always passed as the first argument to the filter function. Custom filters can be configured by registering the Python function as a new filter using the Environment object: def lower_first_filter(s): s = str(s) return s[0].lower() + s[1:] env = Environment(loader=FileSystemLoader(search_path), trim_blocks=True, lstrip_blocks=True ) env.filters['lowerfirst'] = lower_first_filter When a filter named Lower First is available in the template: A list of all supported filters is found as a filter reference. Variables Sample variables are defined by the context dictionary passed to the model. Variables can be complex statements with their own attributes. The entry (.) allows you to use variable attributes in addition to the standard Python __getitem__ syntax ([]). The following lines match: {{ foo.bar }} {{ foo['bar'] }} If there is no variable or attribute, its value will result in an undefined value. The interpretation of this value depends on the configuration of the application: the default action is to evaluate an empty string if it is printed or iterated, and fail all other actions. To comment on a part of a line in a template, use a comment syntax with

{# ... #}by default. This is useful when you want to comment on model elements for debugging or add information to other model designers or yourself: {# note: commented template because we no longer use this {% for users %} ... {% endfor %} #} Line statements If the application enables line statements, the line can be marked as an expression. For example, if the prefix for a wrapping is set to #, the following two examples match: <ul> # for seq <li>{{ destination }}</li># endfor</ul><ul> {% for seq %} <li>{{ item }}</li> {% end %} </ul> The prefix of the line extract can appear anywhere on the line as long as there is no preceding text. To improve readability, the expressions that trigger the block (such as if, if, elif, etc.) can end with a colon: # seq.'s title: ... # The end of line phrases can extend to multiple lines if there are open parentheses, supports, or square brackets: <ul> # href, title [('index.html', 'Index'), ('about.html', 'About')]] <li><a href={{ href }}>{{ caption }}</a></li> # endfor </ul> Since Jinja 2.2, line-based comments are also available. For example, if the prefix for line comments is set to ##, everything from ## to the end of the line is ignored (except for a new line character): # for the item in seq: <li>{{ item }}</li> ## this comment is ignored # endfor Assignmentss code blocks, you you also set values for variables. Top-level assignments (outside blocks, macros, or loops) are exported from a template, such as top-level macros, and can be imported by other templates. Assignments use the specified tag and can have multiple objectives: {% set navigation = [('index.html', 'Index'), ('about.html', 'About')] %} {% set key, value = call_something() %} Variables cannot be set inside the block and can make them appear outside it. This also applies to loops. The only exception to this rule is if statements that do not have a scope. Space control In the default configuration: one new line is removed if it has other spaces (spaces, tabs, new lines, etc.) restored unchanged If the application designates Jinja as trim_blocks, the first new line after automatically removing the model identifier (as in PHP). Lstrip_blocks can also be set as tape tabs and spaces from the beginning of the line to the beginning of the block. (Nothing is removed if there are other signs before the start of the block) When both trim_blocks and lstrip_blocks enabled, you can put block tags on their own line, and the entire block row is removed rendered while preserving the spaces in the content. For example, if you do not have trim_blocks lstrip_blocks settings, this template: <div> {% if true %} yay {% endif %} </div> Rendered with empty lines inside the div: But when both trim_blocks and lstrip_blocks are enabled, the rows in the model block are deleted and other spaces are preserved: the Lstrip_blocks function can be manually disabled by placing a plus sign (+) at the beginning of the block: <div> {%+ if anything like %}yay{% endif %} </div> It is also possible to remove spaces from models. When there is a minus sign (-) at the beginning or end of the block (e.g. For tag), comment, or variable expression, pre-block, or later spaces are removed: for {%seq -%} {}- endfor %} This produces all elements with no space between them. If seq were a list of numbers 1-9, output would be 123456789. If line statements are enabled, they automatically delete the space until the beginning of the line. By default, Jinja2 also deletes new lines. To keep individual new lines, specify Jinja keep_trailing_newline. You must not insert a space between the tag and the minus sign. valid: {%- if foo -%}... {% endif %} is invalid: {% - if foo - %}... {% endif %} %}